

Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs

Neha Rungta
Dept. of Computer Science
Brigham Young University
Provo, UT 84602, USA
neha@cs.byu.edu

Eric G. Mercer
Dept. of Computer Science
Brigham Young University
Provo, UT 84602, USA
egm@cs.byu.edu

ABSTRACT

In this work we focus on creating a benchmark suite of concurrent programs for various programming languages to evaluate the bug detection capabilities of various tools and techniques. We have compiled a set of Java benchmarks from various sources and our own efforts. For many of the Java examples we have created equivalent C# programs. All the benchmarks are available for download. In our multi-language benchmark suite we compare results from various tools: CalFuzzer, ConTest, CHESS, and Java Pathfinder. In Java Pathfinder we provide extensive results for stateless random walk, randomized depth-first search, and guided search using abstraction refinement. Using data from our study we argue that iterative context-bounding and dynamic partial order reduction are not sufficient to render model checking for testing concurrent programs tractable and secondary techniques such as guidance strategies are required. As part of this work, we have also created a wiki to publish benchmark details and data from various tools on those benchmarks to a broader research community.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification; D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

General Terms

Empirical Study, Testing and Verification tools

Keywords

Benchmarks, Concurrent programs, Evaluation

1. INTRODUCTION

The last several years have seen growth in both multi-core processors and a desire to write concurrent programs to take advantage of these multi-core processors. The growth,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PADTAD '09, July 19-20, 2009, Chicago, Illinois, USA.

Copyright 2009 ACM 978-1-60558-655-7/09/07 ...\$10.00.

however, has not been matched by any improvement in our ability to test, analyze, and debug concurrent programs. For example, despite the proliferation of concurrent programs, developers are largely unaware as to how these programs should be tested (or even written for that matter), and as a result often employ stress testing which is known to be very ineffective in detecting concurrency errors.

To be fair, the growth in concurrent programming has been matched by research into methods such as model checking to test, analyze, and debug such programs; although, the research has yet to be widely adopted or even shown to be practically applicable to mainstream programming. This assertion we believe to be true of static analysis, test generation, model checking, or any other such technique (otherwise we would be using such a tool rather than researching it). We believe there are several reasons for such a slow uptake such as

- i) No technique has been shown effective in a general setting. Of the myriad of algorithms and tools, there has yet to be a single approach (or even a combination of approaches) that is demonstrated clearly to be effective.
- ii) No clear comparison of competing approaches. It is very difficult to even begin to identify and select a possible superior approach because, as is common in academics and especially model checking, every tool uses a different input language, produces a different output format, and tests on a unique set of benchmarks often distinct from other researchers (and not available too). How is one to compare without re-implementing every published technique that is seemingly useful?
- iii) We have yet to discover the right technology (or combination of technologies). The research community may still be in search of the needed technology to manage, maintain, and develop the emerging concurrent world.
- iv) There is no money in it. Perhaps it is just a matter of capitalistic forces and no group or individual has seen the right technology to produce sufficient revenue to justify and overcome the development costs of producing a useful and effective tool.

Regardless of the reason for the slow uptake of these emerging technologies to support concurrent programming, we as researchers are obliged to be more scientific in our quest to tackle the problem of test, analysis, and debug of such programs. We ought to have a common set of problems for

which we produce results using our various techniques, and the problems in the set are sufficient to rationally compare competing technologies. Such a commonality benefits the researcher, the business person, and the developer as decisions can now be made from a common reference point.

As one of many first steps (by ourselves and other groups) in the direction of commonality, this paper presents our modest efforts to produce a benchmark suite of concurrent programs for multiple programming languages with results from several tools. Although our focus in the benchmark suite is detecting shared memory concurrency errors in the *testing* mindset, such a suite can be used for analysis and debug as well. Our benchmark suite builds on the Dwyer FSE 2006 benchmark suite, [2], and the IBM benchmark suite, [4, 5], by adding new models including C# versions of many of the models. We also follow the pattern of the DiVinE tool from the Paradise labs, [12], in that we publish the results for the benchmarks; only we also include results from other tools. Unlike other efforts, we also make the raw data available for mining, and we have put everything in a public repository where other researchers can contribute as appropriate. To demonstrate the value of such a common reference point, we present a small empirical study on the data we have thus far collected that motivates guidance strategies and randomization to improve error discovery in dynamic software model checking. The main contributions of this paper are

- i) **Multi-language Benchmarks:** we have taken our set of Java benchmarks compiled from academia, IBM, and our own efforts and created many equivalent C# versions. All of the benchmarks are available to download, and we are working on C versions of the same benchmarks using pthreads. Such a multi-language collection of benchmarks is important to understanding and evaluating different technologies for detecting concurrency errors.
- ii) **Multi-tool Results:** for select models in our multi-language benchmark suite we have results for **CalFuzzer**, **ConTest**, **CHESS**, and **Java Pathfinder**. For **Java Pathfinder** we provide extensive results for stateless random walk, randomized depth-first search, and guided search using abstraction refinement. We are working on results for Inspect which is a dynamic model checker for C programs using pthreads. Such a repository of raw data facilitates more rigorous data analysis for future technologies and the needs of other researchers in the area.
- iii) **On-line Resource:** we have created a wiki to publish benchmark details and data from various tools on those benchmarks to a broader research community. Such an on-line publication encourages researchers to compare emergent technologies for detecting concurrency errors to current state of the art. It also identifies the strengths and weaknesses of such technologies.
- iv) **Empirical Support of Randomization and Guidance:** using the data from our study we provide a modest empirical analysis showing the merits of randomization and guidance in improving error discovery in dynamic model checking. We further argue that techniques such as iterative context bounding and dynamic partial order reduction are not sufficient to ren-

der model checking tractable and secondary techniques such as guidance strategies are required if model checking is ever to be practical in mainstream development. Further results to support this claim are found on the on-line resource.

2. BENCHMARKS

A set of 45 unique Java programs has been collected from various sources [5, 21, 2, 15, 18]. The benchmarks encompass a wide variety of Java programs and concurrency errors. Program derived from concurrency literature, small to medium sized realistic programs obtained from **sourceforge**, models developed at IBM to support their analyses research, and programs designed to exhibit patterns of concurrency bugs usually found in real-world programs. The programs have been parameterized in order to control the number of threads in the program. This allows us to study the effectiveness of the error discovery tool or technique as the number of threads increase in the program.

One of the contributions of this work is that we have created a set of C# programs corresponding to many of the Java programs. As of writing we have C# programs for 12 unique Java models. The methods responsible for generating the various threads in the Java programs are used to create unit tests in the C# programs. The unit tests start the appropriate threads needed to execute the concurrent program. The C# programs that are created essentially have the same functionality and behavior as the original Java programs. For these models we also have multi-tool results.

The Java and their corresponding C# programs are available from a public repository. Furthermore, the data generated from the tests is also available. The details of how to obtain the benchmarks and data are available on a wiki location at:

<http://vv.cs.byu.edu> → **Research Projects** → **Concurrency Tool Comparison**.

We aim to continue and grow this repository by adding more examples for Java and C#. Furthermore we will create equivalent C programs that use the pthread API.

3. MULTI-TOOL RESULTS

In this section we present an overview of the various tools and techniques that are evaluated on a set of concurrent programs. The CHESS concurrency testing tool is used to check multi-threaded C# programs while all other techniques and tools are used to check multi-threaded Java programs. The random walk, randomized depth-first search (DFS), and abstraction guided refinement techniques are implemented and evaluated in the Java Pathfinder (JPF) model checker [21] with dynamic partial order reduction turned on [6]. The JPF model checker is a modified Java virtual machine that provides the ability to systematically explore all possible thread schedules in concurrent programs. ConTest and CalFuzzer are dynamic analysis tools that rely on instrumentation to control the scheduling of concurrent programs.

Random Walk: Random walk is a stateless search technique [7, 20]. Starting from the initial state of the program random walk randomly picks one of the possible successors of the current state to explore a sequence of states in the transition graph. The search continues until it either reaches a

state with no successors (end state), an error state, or some user-specified depth-bound. Due to its stateless nature it does not store any information on previously visited states.

Randomized Depth-first Search: A randomized depth-first search (DFS) is a stateful search technique [2, 1]. Similar to random walk, a randomized DFS explores a sequence of states starting from some initial state. At each state it randomly picks one successor to explore in a depth-first manner. In order to obtain better coverage of the transition graph, it maintains a set of *visited* states to track states that have been explored.

CHES with iterative context-bounding: CHES is a concurrency testing tool for C# programs [10, 11]. It systematically explores the various thread schedules *deterministically*. CHES requires an idempotent unit-test that creates the requisite threads to test a piece of concurrent program. In an idempotent test, the number of threads running at the end of the test needs to be the same as the number of threads running before the start of the test. Furthermore all allocated resources are freed and the global state is reset. CHES executes the unit test and explores a different schedule at iteration of the test. CHES is also a stateless search technique that does not track any information on the states.

The iterative context-bounding approach bounds the number of preemptions along a certain path in order to reach the error faster [10]. To further restrict the number of preemptions CHES only considers preemption points simply at synchronization operations. In programs that contain data races, however, CHES provides a knob for turning on preemptions at accesses to shared volatile variables.

Guided Search with Abstraction Refinement: The guided search using abstraction refinement attempts to verify the reachability of a set of target locations that represent a possible error in the program [17]. The input locations are either generated from imprecise static analysis warnings or user-specified reachability properties. An abstract system contains call sites, conditional statements, data definitions, and synchronization points in the program that lie along control paths from the start of the program to the target locations. The program execution is then guided toward the locations in the abstract system in order to reach the target locations. A combination of heuristics are used to automatically pick thread identifiers [15, 16]. At points in the execution when the program execution cannot be guided further along a sequence of locations (e.g. a particular conditional statement) the abstract system is refined by adding program statements that re-define the variables of interest.

Contest Contest is a concurrency testing tool for Java programs [4]. It attempts to insert noise (randomness) at various synchronization operations while dynamically running the program. It uses a variety of heuristics to drive the schedules. It closely resembles random walk in both behavior and technique.

CalFuzzer: There are two parts to the tool. The RaceFuzzer uses an existing hybrid dynamic race detection to compute a pair of program states that could potentially result in a race condition [18]. RaceFuzzer randomly selects a thread schedule until a thread reaches one program location that is part of the potential data race pair. At this point the execution of the thread is paused at that program location, while the other thread schedules are randomly picked in order to drive another thread to the second program location

that is part of the possible data race. When two threads reach the program location part of the data race and access the same memory location such that at least one of the accesses is a write operation then a real data race is reported. Similarly the DeadlockFuzzer uses the Goodlocks algorithm and lockset analysis, [8, 3], to detect potential deadlocks in the program and randomly drives the threads toward the program locations similar to the abstraction guided search in JPF. CalFuzzer, however, uses a naive guidance strategy. It randomly picks thread schedules until the thread reaches a point of interest.

In a manner similar to CHES, CalFuzzer is stateless. Also it only performs thread switches before synchronization operations and input program locations that represent the potential data race in the program. If the reachability of the program locations, however, depends on another data race in the program, the current implementation of CalFuzzer is unable to detect the error. To overcome this limitation the tool has to insert preemption points at *all data races* in the program. The performance of CalFuzzer would most likely degrade considerably, if preemption points were added at all data races, for two reasons: (1) the overhead in the runtime of the analyses used to detect all data races and (2) the increase in the size of the transition graph resulting from the larger number of preemption points in the program.

4. ON-LINE RESOURCE

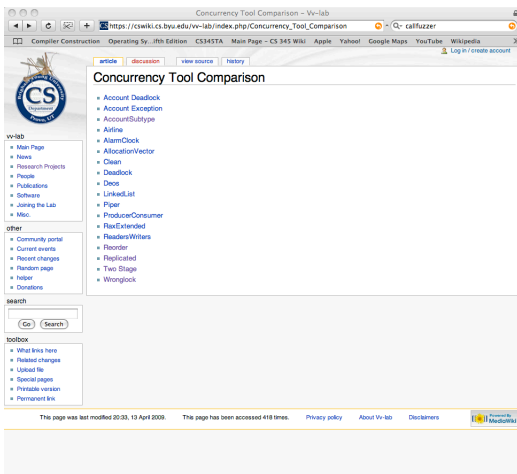
We have created a *concurrency tool comparison* wiki to publish details on each available benchmark including summary tables for results from each of the tools for which the benchmark has been run with tool specific tables showing more complete output. The wiki is located at:

<http://vv.cs.byu.edu> → Research Projects → Concurrency Tool Comparison

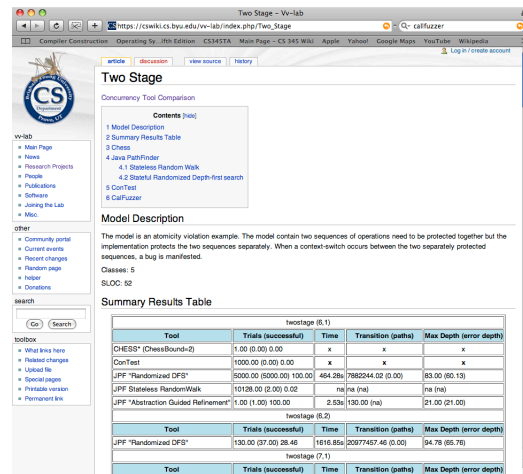
Fig. 1 shows screen shots from the wiki. Fig. 1(a) is the main screen listing benchmarks in the repository. Future work displays the main page as a table providing summary data on the benchmarks including location count, class count, thread count as a function of parameters, error type, and a notion of hardness. Hardness in our work is based on randomized DFS and is a ratio of error finding randomized DFS trials divided by a total number of trials. The idea is that randomized DFS, at a minimum, should fail to find an error most of the time for a model to be considered *hard*.

Fig. 1(b) is an example of the information found on a specific benchmark page. Each benchmark page includes a description with location count, class count, parameters, the number of threads as a function of parameters, and the type of error in the benchmark. It also includes different languages expressing the model. For the shown example, it exists in both Java and C# currently.

After the description comes a complete summary table displaying data from all the tools run on the benchmark. The summary data currently lists the tool, the trials run including successful trials in ()'s with the ratio of successful trials to total trials next to the ()'s (for deterministic tools the number of trials is 1), time in seconds, transition count with paths explored in ()'s, and max depth explored with error depth in ()'s. We are attempting to select summary statistics that would be applicable to most model checkers or systematic search tools; although, as noted in the intro-



(a)



(b)

Figure 1: Concurrency Tool Comparison wiki containing benchmark details with summary multi-tool summary results and tool specific results. (a) Top level page showing the available models. (b) An example of a model page.

duction, there is no general consensus on what should be output from any given tool.

5. EMPIRICAL STUDY

We present a summary of the interesting results by evaluating the various tools and techniques on the set of benchmarks. In this empirical study we present the results on the multi-tools on three unique programs. A more extensive comparison across a larger set of benchmarks is provided on the wiki. We vary model, the independent variable, in our study to evaluate the effectiveness of the error discovery by the different techniques. For a particular model we also vary the number of threads and the kind of threads created by each model.

Time Bound: The randomized DFS trials and the CHESST trials are time bounded at one hour. This time bound was picked to be consistent with other studies [2, 1, 14].

Number of Trials: Over the course of the last three years, we have conducted extensive studies [14, 13] on a set of benchmarks collected from [2]. To recreate the results from the parallel randomized stateful search (PRSS) technique, [1], we ran upto 5000 independent trials of randomized DFS each time bounded at one hour. To compare the random walk results reported in [2] for certain benchmarks we have run upto 10,000 independent random walk trials. For those benchmarks, we present the results based on the data that has already been collected. We have executed between 100 to 1000 trials for the other tools. We believe that this number is sufficient to compare the effectiveness of the different non-deterministic techniques. Increasing the number of trials for these tools is part of ongoing work.

Hardness: We measure the semantic measure of hardness in error discovery as described in [14] in order to evaluate the error discovery abilities of the different tools and techniques. For a given benchmark and technique, the hardness is the ratio of the total number of trials executed over the total number of error discovering trials. Consider, for example, if 100 trials are executed and 50 trials are suc-

cessful in finding an error then the hardness is reported as 0.50 or 50%. Non-deterministic algorithms like randomized DFS and random walk provide a range of values between 0 (hard) and 1 (easy), however, for deterministic algorithms the hardness is either 0 (hard) or 1 (easy).

In this study we compare the error discovery capabilities of stateful as well as stateless techniques. In order to measure the effectiveness of error discovery we measure the number of transitions that were executed before error discovery, the maximum depth of the search, and the depth at which the error was found. Note that some tools such as ConTest and CalFuzzer do not output values for the measures. The number of transitions executed are reported to better compare stateful and stateless techniques.

In Table 1, Table 2, and, Table 3 we present the results for the *reorder*, *twostage*, and *airline* benchmarks respectively. The column labeled **Tool** indicates the tool used to evaluate the benchmark. ConTest is run with the default values where it randomly picks one of its heuristics for improving error discovery. The parameters of CHESST indicate whether volatile variables are monitored ($V=true$) and the preemption bound ($B=num$). Recall that preemptions at volatile variables are turned off by default. The randomized DFS, random walk, and abstraction guided search (Abs. Guided) are executed in JPF with dynamic partial order reduction turned on. The *na* entries in the table indicates that the particular output is *not available* for the particular tool. The *x* legend indicates that none of the trials found an error and, hence, there is no data to report.

5.1 Reorder

The benchmark contains a data race. There is a check whether the data race causes an inconsistency in the data values. When such an inconsistency is discovered an exception is raised. The benchmark contains two kinds of threads: **setter** and **getter** that cause a data race in the program. The **getter** thread also checks for the inconsistency in the data values caused by the data race. As we increase the number of **setter** threads while keeping the **getter** thread

Tool	Trials (Successful)	Error Dis. Rate	Time	Transition	Max Depth (Error Depth)
Reorder(2,1) ThreadNum=4					
CalFuzzer	100 (12)	12%	1.376s	na	na(na)
CHESS* (V=true, B=2)	1 (1)	100%	0.44s	2600	26 (na)
ConTest	1000 (23)	2.30%	na	na	na(na)
Randomized DFS	100 (100)	100%	0.61s	560.58	28.33 (21.24)
Random Walk	1000 (2)	0.20%	0.28s	27.50	27.50 (27.50)
Abs. Guided	1 (1)	100 %	1.67s	44	14 (14)
Reorder(1,5) ThreadNum=7					
CalFuzzer	100 (24)	0.24%	1.658s	na	na(na)
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
ConTest	1000 (285)	0.28%	na	na	na(na)
Randomized DFS	100 (100)	100%	16.61s	238450.80	52.94 (25.52)
Random Walk	10128 (438)	4.32%	0.35s	32.41	32.41 (32.41)
Abs. Guided	1 (1)	100%	2.67s	29	12 (12)
Reorder(4,1), ThreadNum=6					
CalFuzzer	100 (12)	0.12%	1.45s	na	na(na)
CHESS* (V=true, B=2)	1 (1)	100	140.53s	1200000	40
ConTest	1000 (3)	0.3%	na	na	na(na)
"Randomized DFS"	100 (100)	100%	7.35s	33895.58	46.61 (34.29)
Random Walk	1000 (0)	0%	x	x	x
Abs. Guided	1 (1)	100%	1.67s	80	18 (18)
Reorder(9,1) ThreadNum=11					
CalFuzzer	100 (10)	0.10%	1.49s	na	na(na)
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
ConTest	1000 (10)	0.01%	na	na	na(na)
Randomized DFS	5000 (593)	11.86%	2497.98s	38704112.56	81.67 (59.64)
Random Walk	10124 (0)	0%	x	x	x
"Abs. Guided"	1 (1)	100%	1.67s	205	28 (28)
Reorder(10,1) ThreadNum=12					
CalFuzzer	100 (9)	0.09%	1.49s	na	na (na)
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	5000 (4)	0.08%	2414.82s	38022689.25	89.75 (65.25)
Random Walk	10127 (0)	0%	x	x	x
Abs. Guided	1 (1)	100%	1.67s	236	30 (30)

Table 1: Comparing the error discovery of different techniques on the reorder benchmark.

constant, the semantic measure of the hardness in error discovery increases (i.e., get harder) as shown in [14].

CHES is effective for error discovery in the `reorder` model with a small number of threads as shown in Table 1. The data race in the model can be manifested with a preemption bound of two ($B=2$). We also monitor volatile variables ($V=true$). As the number of threads increases even with a preemption bound of two, CHES progressively takes more time to find the error. In Table 1 we notice that CHES is unable to find an error in `reorder(9,1)` and `reorder(10,1)` in a time bound of one hour.

The error discovering capability of randomized DFS search degrades when the number of threads are increased. In the `reorder(10,1)` model only 8 trials out of 5000 were able to find an error and, on average, took 2414.82 seconds to find an error as shown in Table 1. Random walk and ConTest are only able to find an error in `reorder(2,1)` and `reorder(1,5)` models.

CalFuzzer and the abstraction guided search are most effective in finding errors as the number of threads increase in the `reorder` model. CalFuzzer is more effective than randomized depth-first search because it only considers preemption points at the pair of program locations that are reported to be part of a data race. In the `reorder` model there are no synchronization operations, hence, for CalFuzzer there only exist two points of preemption. The abstraction guided search is able to find the error quickly even when the model checker considers preemption points at all shared variable accesses.

5.2 TwoStage

The benchmark contains an atomicity violation. The program has sequences of operations that need to be protected together but the implementation incorrectly protects the two sequences separately. The input parameters to the benchmark are two different types of threads (`twoStagers` and `readers`). The `twoStage` thread modifies the two separately protected sequences. When the `reader` thread reads the value of a shared variable between the two write accesses by the `twoStage` thread then a bug is manifested (an exception is raised).

CalFuzzer is unable to find a set of input location using their initial dynamic analysis. In the current configuration of the tool we are unable to manually specify the input location to CalFuzzer. The input target location is the program location where the exception is raised. This is to check whether the exception is ever raised. The ConTest tool is unable to find the bug in 1000 trials for any configuration of the model. Random walk is also not very effective in error discovery. The error discovery rate for `twostage(1,1)` is only 4.25%.

CHES is effective in error discovery for the `twostage(1,1)` and `twostage(1,2)` models where there are a small number of threads. The `twostage(1,1)` has 3 total threads while `twostage(1,2)` has 4 threads. As, however, the number of threads in the benchmark increase, CHES is less effective in error discovery. Note that in `twostage(2,2)` the average time taken by randomized DFS to find the error is only 2.41 seconds while CHES takes 17.44 seconds to find the error.

The abstraction guided search does not degrade as we increase the number of threads in the program. So while it takes more time to detect the error in the benchmark when there is a small number of threads (such as `twostage(1,1)`)

compared to the other techniques, the time stays consistent even as we increase the number of threads. In benchmarks where increasing the number of threads does not change the conditions that cause the error, the abstraction guided search is consistent in its performance showing little change in running time.

5.3 Airline

The benchmark contains a data race. As the number of threads are created the value of a global variable is updated. An incorrect assumption on atomicity allows more threads to be created leading to a data inconsistency. The two parameters to the `airline` model are: `ticketsIssued` and `cushion`. The minimum depth of the error is pushed deeper in the execution trace when we increase the value of the `cushion` and keep the total number of threads, `ticketsIssued`, constant.

The reachability of the error state in the `airline` model depends on the value of a global counter that is modified by different threads. Based on the input parameters of the model, a different number of preemptions is required to elicit the error in the airline model. In Table 3 for the `airline(20,1)` model, 18 preemptions are required to elicit the error. Even after setting the correct number of preemptions, CHES is unable to discover the error in a time bound of one hour. The need to specify the correct preemption bound is another limitation of the iterative context-bounding approach.

Recall that CalFuzzer only inserts preemption points at synchronization points and the input pair of program locations that represent a potential data race in the program. In the `airline` model, CalFuzzer is unable to find the error in the benchmark because a specific number of preemptions at unprotected global variables are required to elicit the bug.

The abstraction guided search technique can successfully find the error in the model. In the refinement process the location of setting the global variable is iteratively (and automatically) added. This enables the abstraction guided search to find the error even when a specific sequence of preemptions are required. For example, `airline(20,1)` requires the most number of refinements, hence, its total time for error discovery is 7.46 seconds while the model `airline(4,1)` that requires fewer refinements takes only 3.46 seconds. The results are encouraging since the other techniques struggle to find an error in the model within significant constraints of time and memory.

5.4 Discussion

The performance of CHES is hindered due to its deterministic nature. Even in the presence of iterative context-bounding CHES is severely limited by the benefits and limitations of default search order. With a systematic randomization implemented within CHES, we believe it can outperform randomized DFS in models that require only one or two preemptions along an execution path to reach an error location. In large programs, however, where the errors exist along very few paths in the transition graph, iterative-context bounding is not likely to pick the schedule that leads to an error state (even with randomization).

The performance of ConTest is comparable to that of random walk. A more systematic random search such as randomized depth-first search is in general more effective for error discovery within the same constraints of time and memory. At a certain point, however, simple randomized search

Tool	Trials (Successful)	Error Dis. Rate	Time	Transition	Max Depth (Error Depth)
TwoStage(1,1), ThreadNum=3					
CHES* (B=2)	1 (1)	100%	0.94s	180	20 (na)
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	0.61s	88.04	20.18 (18.51)
Random Walk	10127 (430)	4.25%	0.02s	0.94	0.94 (0.94)
Abs. Guided	1 (1)	100%	3.53s	30	11 (11)
TwoStage(1,2), ThreadNum=4					
CHES* (B=2)	1 (1)	100%	0.52s	5800	29 (na)
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	1.07s	1966.10	38.46 (30.08)
Random Walk	1000 (10)	1%	0.30s	30.20	30.20 (30.20)
Abs. Guided	1 (1)	100%	3.53s	46	13 (13)
TwoStage(2,2), ThreadNum=5					
CHES* (B=2)	1 (1)	100%	17.44s	228000	38 (na)
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	2.41s	10223.50	45.72 (36.07)
Abs. Guided	1 (1)	100%	3.53s	46	13 (13)
TwoStage(1,5), ThreadNum=7					
CHES* (B=2)	1 (0)	0%	x	x	x
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	319.01s	3341784.11	73.96 (38.09)
Abs. Guided	1 (1)	100%	2.53s	30	11 (11)
TwoStage(8,1), ThreadNum=10					
CHES* (B=2)	1 (0)	0%	x	x	x
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	4999 (126)	2.52%	2155.76s	32969192.09	105.46 (73.40)
Random Walk	10200 (0)	0%	x	x	x
Abs. Guided	1 (1)	100%	2.53s	184	25 (25)

Table 2: Comparing the error discovery of different techniques on the twostage benchmark.

Tool	Trials (Successful)	Error Dis. Rate	Time	Transition	Max Depth (Error Depth)
Airline(4,1), ThreadNum=5					
CalFuzzer	100 (0)	0%	x	x	x
CHES* (V=true, B=2)	1 (1)	100%	231.240s	5,200,425	114 (na)
CHES* (V=true, B=3)	1 (1)	100%	1982.522s	44,916,000	114 (na)
Randomized DFS	200 (200)	100%	0.81s	5776.74	14.28 (13.23)
Random Walk	1011 (98)	9.69%	0.25s	22.85	22.85 (22.85)
Abs. Guided	1 (1)	100%	3.46s	61	17 (17)
Airline(4,2), ThreadNum=5					
CalFuzzer	100 (0)	0%	x	x	x
CHES* (V=true, B=2)	1 (1)	100%	67.533s	1,482,000	114 (na)
Randomized DFS	200 (200)	100%	0.25s	655.86	13.75 (12.59)
Random Walk	1021 (433)	42.41%	0.19s	19.05	19.05 (19.05)
Abs. Guided	1 (1)	100%	3.46s	46	14 (14)
Airline(5,1), ThreadNum=7					
CalFuzzer	100 (0)	0%	x	x	x
CHES* (V=true, B=2)	1 (0)	0%	x	x	x
CHES* (V=true, B=3)	1 (0)	0%	x	x	x
Randomized DFS	200 (200)	100%	58.19s	670929.49	25.27 (22.17)
Random Walk	1021 (111)	10.87%	0.39s	28.75	28.75 (28.75)
Abs. Guided	1 (1)	100%	3.46s	98	21 (21)
Airline(20,8), ThreadNum=21					
CalFuzzer	100 (0)	0%	x	x	x
CHES* (V=true, B=2)	1 (0)	0%	x	x	x
Randomized DFS	5000 (2507)	50.14%	285.85s	4111966.14	121.07 (118.62)
Random Walk	10123 (696)	6.88%	0.09s	12.98	12.98 (12.98)
Abs. Guided	1 (1)	100%	5.46s	2680	60 (60)
Airline(20,3), ThreadNum=21					
CalFuzzer	100 (0)	0%	x	x	x
CHES* (V=true, B=2)	1 (0)	0%	x	x	x
Randomized DFS	4992 (24)	0.48%	375.78s	5905884.67	121.75 (119.75)
Random Walk	11132 (19)	0.17%	0.44s	110.63	110.63 (110.63)
Abs. Guided	1 (1)	100%	7.46s	3210	75 (75)
Airline(20,1), ThreadNum=21					
CalFuzzer	100 (0)	0%	x	x	x
CHES* (V=true, B=18)	1 (0)	0%	x	x	x
Randomized DFS	4996 (0)	0%	x	x	x
Random Walk	11125 (0)	0.00%	x	x	x
Abs. Guided	1 (1)	100%	7.46s	3609	95 (95)

Table 3: Comparing the error discovery of different techniques on the airline benchmark.

techniques fail to find an error. The data from our empirical study demonstrate that iterative context-bounding and dynamic partial order reduction are not sufficient to find errors in concurrent programs. There is a need for a more sophisticated guidance such as the guided search with abstraction refinement that consistently performs well across the different benchmarks in this study.

6. RELATED WORK

Various efforts have been made to compile a set of benchmarks for concurrent programs [4, 5]. The programs collected exhibit a variety of concurrency errors. The benchmark suite has multi-threaded programs with documented bugs. The annotations about bugs in the program also helps evaluate imprecise static and dynamic analysis technique in determining whether the warning on a possible error is a false positive. This work attempts to take the idea of a benchmark suite one step further by having multi-language programs and results from different tools on the programs.

The BEEM—Benchmarks for Explicit Model Checkers is a benchmark set that contains 50 parameterized models along with their correctness properties [12]. For a given model, an instance generator can generate models in the DiVinE specification language (DVE) as well as Promela, the input language for the SPIN model checker [9]. Our work here attempts to provide the same resource for programs targeted to software model checking rather than pure model checking.

The work on defining a semantic hardness measure for concurrent programs using randomized depth-first search has allowed us to test new techniques on models where exhaustive and randomized techniques fail to find an error [14, 15, 17]. Our experience shows that the hardness measures generated by a randomized depth-first search with partial order reduction turned on provides a good threshold that needs to be overcome by a new technique or tool in order for that technique or tool to be considered effective in error discovery.

7. CONCLUSION

We present in this work a modest attempt to bring commonality and a sure reference point to research for the test, analysis, and debug of concurrent programs. Our contribution is in the form multi-language benchmarks, multi-tool results, an on-line resource for broader impact, and an empirical study showing the merits of various techniques. All of the models are open to other researchers, and we hope other are able to contribute to the work. An example of the effectiveness of such a common reference for research is in our empirical study. Using the data from our study show that techniques such as iterative context bounding and dynamic partial order reduction are not sufficient to render model checking tractable and secondary techniques such as guidance strategies are required if model checking is ever to be practical in mainstream development.

We intend to continue to publish benchmarks and data on the wiki. Immediate future work is to automate table construction from the raw data using various scripts. We are also working on a summary table showing all the available benchmarks as the opening main page rather than the simple list of benchmarks. We hope to continue to produce C# models as appropriate. The next target tool is Inspect

requiring C pthread models. We are also looking to begin exploring and publishing results from automated test tools like jCUTE [19]. jCUTE is a concolic testing framework for concurrent programs using a dynamic partial order reduction.

8. ACKNOWLEDGEMENTS

We would like to thank Ira and Mary Lou Fulton for their generous donations to the BYU SuperComputing laboratory which made it possible for us to run the extensive analyses presented in this paper and online. We would also like to thank Kevin Seppi and Andrew McNabb for the use of the computing cluster in the Applied Machine Learning Lab at BYU.

9. REFERENCES

- [1] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
- [2] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 92–104, New York, NY, USA, 2006. ACM Press.
- [3] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proc. SOSP '03*, pages 237–252, New York, NY, USA, 2003. ACM Press.
- [4] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):267–279, 2007.
- [5] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, page 266a, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [6] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL*, pages 110–121, New York, NY, USA, 2005. ACM.
- [7] P. Haslum. Model checking by random walk. In *Proceedings of ECSEL Workshop*, 1999.
- [8] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proceedings of the 7th International SPIN Workshop on Software Model Checking*, pages 245–264, London, UK, 2000. Springer-Verlag.
- [9] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [10] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.
- [11] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proc. of PLDI*, pages 362–371, New York, NY, USA, 2008. ACM.
- [12] R. Pelanek. BEEM: Benchmarks for explicit model checkers. *Lecture Notes in Computer Science*, 4595:263, 2007.

- [13] N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In C. S. Pasareanu, editor, *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, pages 39–57, Berlin, Germany, July 2007. Springer–Verlag.
- [14] N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *5th IEEE International Conference on Software Engineering and Formal Methods*, pages 247–256, London, U.K, September 2007.
- [15] N. Rungta and E. G. Mercer. A meta heuristic for effectively detecting concurrency errors. In H. Chockler and A. J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2008.
- [16] N. Rungta and E. G. Mercer. Guided model checking for programs with polymorphism. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 21–30, New York, NY, USA, 2009. ACM.
- [17] N. Rungta, E. G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proceedings of the 16th International SPIN Workshop on Model Checking of Software*, pages 174–191, Grenoble, France, June 2009. Springer–Verlag.
- [18] K. Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008.
- [19] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proc. HVC*, volume 4383 of *LNCS*, pages 166–182. Springer, 2007.
- [20] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.
- [21] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. ASE*, Grenoble, France, September 2000.